Kristen Harrison
CS162, Project 1: Langton's Ant

Reflections document to include the design description, test plan, test results, and comments about how you resolved problems during the assignment

Design description:

Assignment requirements – The program implements a simulation of Langton's Ant. It accepts user input to set the number of rows and columns, the number of steps the ant takes, and the starting coordinates of the ant. Langton's Ant involves an algorithm by which the ant, if on a white square, turns right 90 degrees and flips the square to black, and if on a black square, turns left 90 degrees and flips the square to white. The ant then steps forward. At each step the program needs to print out the board, and a menu is required to let the user choose a version of the game.

To implement this, I created two classes: Board and Ant, with Board as the parent class, which has an Ant object. A board is created with 4 parameters: the number of rows and columns on the board, and the starting row and column for the ant object.

Board board = Board(numRows, numCols, startRow, startCol);

The Board constructor initializes those variables as data members, creates a dynamically allocated array with the number of rows and columns specified,

grid = new char*[numRows]; (for loop from 0 to i < numRows) {grid[i] = new char[numCols];}

and then creates its Ant object with two parameters: just the starting row and column of the ant.

ant = Ant(startRow, startCol);

Functions of Board include moveAnt(), which controls the overall movement on the board, and print():

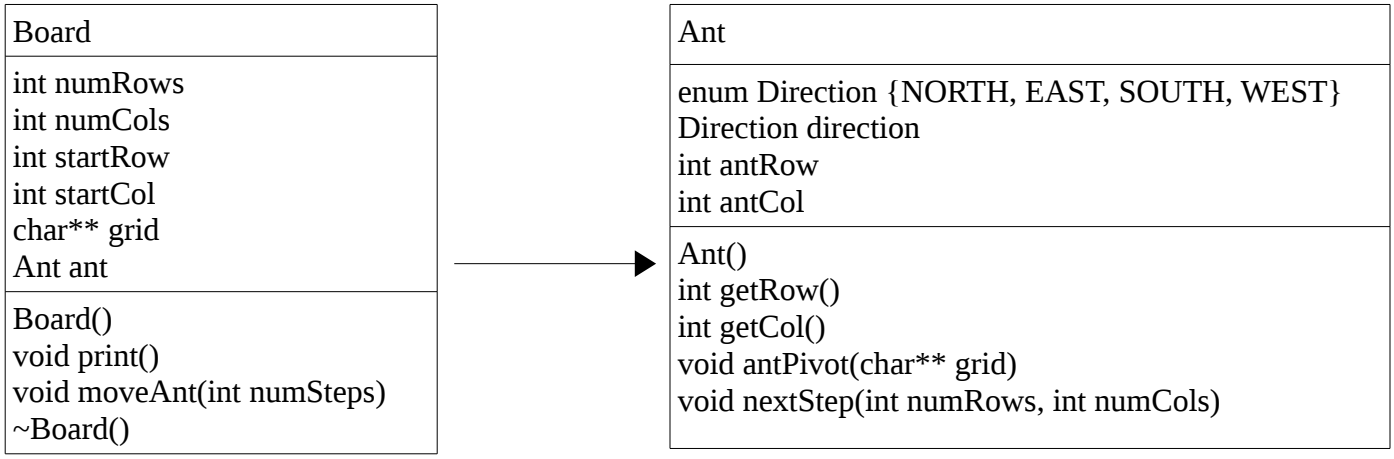
moveAnt(numSteps); use loop to count to number of moves

calls print(); ant.antPivot(grid); ant.nextStep(numRows, numCols);

print() loops through the array with two for-loops to print out the arrays contents, plus formatting (not shown here).

```
for i from 0 to numRows
        for j from 0 to numCols
                if i == ant.getRow() && j == ant.getCol()    {cout '@'}
                else     {cout 'grid[i][j]}
```

The Board destructor frees memory by using a for loop from 0 to numRows to delete all the dynamically allocated arrays and set the pointers to null.



Board has an Ant, which it constructs by passing Ant(startRow, startCol), which are stored as antRow and antCol. Ant also has its own enum called Direction, which keeps track of the direction the ant faces. The ant is controlled by Board's moveAnt function, which loops though numSteps to call ant.antPivot and ant.nextStep. Ant function algorithms are centered on the numbers associated with the cardinal directions in the enum.

antPivot() direction is added to turn right (unless already 3), and subtracted to go left (unless 0):

if grid[antRow][antCol] == ' ' (go right)

{ if dir < 3, dir ++; else dir = 0; grid[antRow][antCol] = '#' }

if grid[antRow][antCol] == '#' (go left)

{ if dir > 0 dir--; else dir = 3; grid[antRow][antCol] = ' ' }

nextStep() the algorithm checks the direction, then increments or decrements a row or column:

if dir == NORTH

if row > 0 {row--}; else row = numRows - 1;

else if dir == EAST

if col < numCols – 1 {col++} else col = 0;

else if dir == SOUTH

if row < numRows – 1 {row++} else row = 0;

else if dir == WEST

if col > 0 {col--}else col = numCols – 1;

The algorithm checks if the ant is going over the edge of the board, and if so, wraps it around to the other side.

Test cases:

Most of my test cases relied on input validation functions. In order to keep this concise, I'm omitting all the repeats that involved sending incorrect input back to the menu, because they had the expected feedback. I tested the menu at each level that required user input, and everything that required an int within a certain range accurately denied any letter, special character, whitespace, float, or integer outside the range specified. I expected "Invalid input, please enter an integer between [min] and [max]" until correct input was supplied, which was what happened for each one.

My main issue was working out class inheritance and using enums and the related syntax. Once I had my Board and Ant classes working, most of the rest was fine. It's actually awkward to try to explain this with a test table, because I did not have any issues with my function algorithms working and it was not the user input that caused any problems. Everything that broke was syntax related, and it's difficult to express that here in a table. My table would either show that almost all test cases worked (once I fixed the syntax), or that every "Actual result" was a compiler error.

In the interest of making this make sense, these are the three problems that I found through testing input/output, and the other problems I will address in the final "what did I have to change" section.

input	**expect output**	**actual output**	**notes**
0 rows in the array	the menu again, because the print() loop would be going from 0 to 0, so would print nothing	error messages from input validation, asking me to input an integer between 0 and -1 for the ant starting row	I fixed the code to require at least 1 row and column
numRows = 8 numCols = 5 numSteps = 10 startRow = 8 startCol = 2	ant starts in row 8, column 2	no ant in sight segmentation fault	"off by one" error. I had to change the acceptable range for startRow (and startCol) to (0, numRow - 1)
numRows = 80 numCols = 80 numSteps = 20 random assignments for startRow and startCol	an array the width of the terminal	a total mess	I had forgotten that with a border of asterisks around the array, it couldn't be any wider than 78 without overflowing. I set the max to 70

What had to change:

The algorithms I drew up to handle individual ant movements – pivoting left or right, and moving forward one step – seemed to work well, and I didn't change them during implementation. I made drastic changes to class organization, though. I originally had Ant as the parent class, with a Board

object, because when I was looking through past assignments, a TicTacToe game had used this approach and it seemed to make sense.

But as I was fleshing out the distribution of functions, it felt lopsided because the only thing Board had to do was print out the array to the console. That seemed like a waste of a class, and I read on the Langton's Ant page that some variations employ multiple ants. The point of modularity is to be able to easily reuse and scale up, and it seemed much more scalable to have Board > Ant instead. An Ant would never need multiple Boards, but a Board could have a use for multiple Ants.

I switched inheritance during implementation, then, and I think the flow makes more sense this way because now Board operates the controlling mechanism of the game, through moveAnt(numSteps), by calling Ant class functions on the Board's Ant object. The Ant class is responsible for individual ant movements, while Board plays the puppetmaster, and could be easily modified to do so for more complex scenarios.

My other edits were syntactical. Using an enum worked better than I expected, but I still had to change how it was written inside antPivot(), because direction++ and direction=0 didn't work as I had hoped. I relearned that you can't perform integer operations on an enum, even though you can compare it to one. I had to change what I had written in my plan to the more awkward and unsightly

direction = static_cast<Direction>(direction + 1); and direction = NORTH;

The rest of the problems were careless oversights on my part: I had originally looped though numCols first rather than numRows in one of my double for loops, and got a segmentation fault when it went out of bounds. I also accidentally left some integer constants outside of the function that used them, which was an easy fix. I hadn't originally written in empty constructors for the classes yet when the Ant class complained about not having one. I ended up assigning default values to the parameters passed to the Board and Ant constructors, because if I'm understanding this correctly, default values enables the class to use it as a default constructor.

The first time I tried to print out the ant icon, it went where it was supposed to but then didn't move from that spot. I hadn't thought much about my print function, because it was one of the smaller details of the project, but when the ant was frozen on the board, I saw of course that I couldn't just assign grid[antRow][antCol] = '@'; I hadn't thought that particular assignment through about how it wouldn't then be able to test whether the square was white or black. So the '@' substitution had to happen within the printing loop rather than actually putting ant on the board.

I also had to make Ant getRow() and getCol() accessor functions on the fly, because since Board prints the array it wasn't able to access the private data member ant.antRow. I knew this, but didn't remember until I was reminded of it by the compiler. Likewise, I later had to amend the parameters of antPivot(grid) and nextStep(numRows, numCols) to pass in the array pointer and the board dimensions because the ant object didn't have access to private Board variables.

I didn't plan out how int main and the menu would work until after the rest of the program was written, because they were fairly standalone, and I wanted to get a functioning Ant and Board first. I didn't have any significant problems with the menu because I made a very simple one: one function, in the utilities library, that prints out the options and then demands an integer response within the proper range. Main, however, I changed significantly to get the menu to repeat how I wanted. Because running the program would be so repetitive, I created a function called runGame and moved into it all the code that would be run once a user made a choice what version of the game they wanted. The main function has only a do-while loop to handle offering the menu, storing the validated choice in int progVersion, and calling runGame(progVersion) inside the loop until the user finally decides to quit.